(1)

David Taylor Research Center

Bethesda, Maryland 20084-5000

AD-A218 917

SSPD-90-175-11 November 1989

Ship Structures and Protection Department

Research and Development Report

VECTORS AND MATRICES: TWO TURBO PASCAL UNITS FOR FAST PROGRAM DEVELOPMENT

by

Peter N. Roth

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

90 03 05 115

SSPD-90-175-11 VECTORS AND MATRICES: TWO TURBO PASCAL UNITS FOR FAST PROGRAM DEVELOPMENT

MAJOR DTRC TECHNICAL COMPONENTS

CODE 011	DIRECTOR OF TECHNOLOGY, PLANS AND ASSESSMENT
12	SHIP SYSTEMS INTEGRATION DEPARTMENT
14	SHIP ELECTROMAGNETIC SIGNATURES DEPARTMENT
15	SHIP HYDROMECHANICS DEPARTMENT
16	AVIATION DEPARTMENT
17	SHIP STRUCTURES AND PROTECTION DEPARTMENT
18	COMPUTATION, MATHEMATICS & LOGISTICS DEPARTMENT
19	SHIP ACOUSTICS DEPARTMENT
27	PROPULSION AND AUXILIARY SYSTEMS DEPARTMENT
28	SHIP MATERIALS ENGINEERING DEPARTMENT

DTRC ISSUES THREE TYPES OF REPORTS:

1. **DTRC reports, a formal series,** contain information of permanent technical value. They carry a consecutive numerical identification regardless of their classification or the originating department.

2. **Departmental reports, a semiformal series,** contain information of a preliminary, temporary, or proprietary nature or of limited interest or significance. They carry a departmental alphanumerical identification.

3. **Technical memoranda, an informal series,** contain technical documentation of limited use and interest. They are primarily working papers intended for internal use. They carry an identifying number which indicates their type and the numerical code of the originating department. Any distribution outside DTRC must be approved by the head of the originating department on a case-by-case basis.

David Taylor Research Center

Bethesda, Maryland 20084-5000

SSPD-90-175-11

November 1989

VECTORS AND MATRICES: TWO TURBO PASCAL UNITS FOR FAST PROGRAM DEVELOPMENT

by

Peter N. Roth

APPROVED FCR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

CONTENTS

ABSTRACT 1
ADMINISTRATIVE INFORMATION 1
1. INTRODUCTION 1
1.1 Pascal in general 1
1.2 Turbo Pascal in particular 1
2. USING THE UNITS 3
3. NOTES ON THE MATRICES UNIT 4
3.1 A Warning 4
3.2 Mechanisms of the Matrices Unit 4
3.3 Using the Unit with Various Size Matrices 4
4. USING THE APPENDICES 6
4.1 How to find a procedure 6
4.2 Conventions in the code 6
5. CONCLUDING REMARKS 7
6. REFERENCES 7
APPENDIX A - PROCEDURE DESCRIPTIONS 8
APPENDIX B - VECTOR CALLING SEQUENCES 9
APPENDIX C - MATRIX CALLING SEQUENCES 10
APPENDIX D - VECTOR EXAMPLES 12
APPENDIX E - MATRIX EXAMPLES 13
11.1 Integer Examples 13
11.2 Float Examples 13
APPENDIX F - THE VECTORS UNIT SOURCE CODE 15
APPENDIX G - THE MATRICES UNIT SOURCE CODE 19

ABSTRACT

Procedures and functions for vector and matrix mathematics in **Turbo Pascal**© are presented.

1. INTRODUCTION

Vector and matrix methods in general, and the finite element method in particular, are now common engineering tools. Vectors are used to represent coordinates of points in 3-space, forces, displacements, velocities, and so forth. Matrices are typically used to represent finite element stiffness or flexibility, damping, mass, coordinate transformation rules, etc. Any modern college mathematics book may be consulted for vector and matrix theory.

This report documents a set of vector and matrix arithmetic procedures in Turbo Pascal. Use of the procedures permits fast program development.

1.1 Pascal in general

Pascal, a programming language invented in the early 1970's by Niklaus Wirth [1], differs from FORTRAN in the way in which parameters are passed to procedures.

In generating the call to a procedure, FORTRAN compilers create a list of the addresses of the parameters. Because each procedure is independent of all others, FORTRAN has no way to ensure that the type of the actual argument matches the type of the "dummy" argument. This powerful language feature can be exploited by careful programming; for example, a single FORTRAN matrix multiply routine may be written to handle arrays of any size. This powerful language feature is also a great source of many illusive bugs, because FORTRAN allows an integer to be passed to a procedure that expects a real, or a scalar variable to a procedure expecting an array, etc.

Pascal "fixes" this FORTRAN problem by ***strong type-checking:*** that is, the compiler ensures that the types of parameters ***passed*** to procedures and functions are in agreement with the types of parameters ***expected*** by them. The Pascal compiler thus trades off "the capability of writing general purpose procedures" for "protection of the programmer from himself". In standard Pascal, a unique multiply procedure is required for each product of different sizes of matrices. This drawback is one of the reasons why Pascal is preferred as an "instructional" language, and explains why it has not replaced FORTRAN.

1.2 Turbo Pascal in particular

Version 5.0 of the Turbo Pascal compiler, manufactured by Borland International [2], [3] extends standard Pascal in several ways. Those extensions that are germane:

- Wirth's Modula-2 [4] ***module***† concept is implemented. This allows separate compilation of code modules while retaining the strong type-checking of Pascal.
- the reserved word **absolute** allows the "equivalencing" of variables;
- the ***untyped*** parameter in procedure calls allows programmers to avoid the strong type-checking of Pascal.

† Borland's spelling for the word "module" is "Unit."

These extensions permit development of general-purpose modules, while still offering many of the advantages of Pascal. They do require increased discipline and vigilance by programmers, however.

The two Turbo Pascal Units documented in this report are **vectors** and **matrices.** These units can be used to get a program running *cleanly* and *correctly* before "efficient" code is attempted. Note that attempts at speed usually require customization of the code to the target machine, with an attendant increase in code entropy (i.e., chaos). It is also important to observe that Turbo Pascal is *not* Pascal: Turbo Pascal is "Pascal-like"; it is not, in general, portable to machines other than PCs.

2. USING THE UNITS

The use of the **vectors** and **matrices** units is easily demonstrated with a typical program:

```
1   program typical ;
2
3   (*. typical -  a typical Turbo Pascal program. *)
4
5   {$I float.inc }
6
7   Uses vectors
8      , matrices
9      ;
10
11  TYPE
12    mat3x4 : array [ 1 .. 3, 1 .. 4 ] of float ;
13    mat4x5 : array [ 1 .. 4, 1 .. 5 ] of float ;
14    mat3x5 : array [ 1 .. 3, 1 .. 5 ] of float ;
15    coordinate_type : array [ 1 .. 100 ] of vector ;
16
17  VAR
18     A : mat3x4 ;
19     B : mat4x5 ;
20     C : mat3x5 ;
21
22     x, y, z : vector ;
23     coordinate : coordinate_type ;
24
25  begin (* typical *)
26     { initialize A and B, x and y, etc. }
27
28     mfmult( C, A, B, 3, 4, 5 ) ;  (* matrix multiply *)
29     vcross( z, x, y ) ;           (* vector cross product *)
30
31  end  (* typical *).
```

Matrices are completely general; their sizes are left up to the programmer. A more detailed discussion of how the **matrices** unit works is given in the next section.

Vectors, on the other hand, are defined in the **vectors** unit:

```
TYPE
  vector = array [ 1 .. 3 ] of float ;
```

so that the type *vector* requires no user declaration.

To allow the accuracy of any given application to be changed to suit, the file <float.inc> defines the accuracy of floating point computations (Turbo Pascal allows five floating point types).

```
TYPE
  float = double ;       (* change to suit accuracy *)
```

This declaration is "included" at line 5 of the program with the compiler directive

```
{$I float inc}
```

3. NOTES ON THE MATRICES UNIT

3.1 A Warning

Programmers using the **matrices** unit must ensure that the proper parameters are passed to the procedures, since type-checking is bypassed!

3.2 Mechanisms of the Matrices Unit

The unit works by matching the addresses of the procedure parameters with its own internal notion of an array. The external definitions are the responsibility of the programmer.

The internal definitions are

```
const
  MAX_FLOAT_MATRIX = 65520 div Sizeof( float ) ;
  MAX_INT_MATRIX   = 65520 div Sizeof( integer ) ;

type
  float_matrix   = array [ 1 .. MAX_FLOAT_MATRIX ] of float ;
  integer_matrix = array [ 1 .. MAX_INT_MATRIX   ] of integer ;
```

The parameters are "matched" via the Turbo Pascal **absolute** reserved word. For example, the procedure *mfadd:*

```
(*****
 *
 *.  mfadd - float matrix addition: [c] := [a] + [b].
 *
 *)
procedure mfadd ( VAR c, a, b ;  n: integer ) ;

  VAR
    cc : float_matrix  absolute  c ;
    aa : float_matrix  absolute  a ;
    bb : float_matrix  absolute  b ;

  begin (* mfadd *)

    for  n := 1  to  n  do
       cc[n] := aa[n] + bb[n]

  end (* mfadd *) ;
```

The external arrays are *a*, *b*, and *c*. All the operations are performed on the internal arrays *aa*, *bb*, and *cc*, each of whose starting addresses is equated to the appropriate external array.

NOTA BENE: The *n* passed to this procedure must not be larger than the defined lengths for *a*, *b*, and *c* or the procedure will access addresses of the machine that are outside the arrays. Results in this case are unpredictable.

3.3 Using the Unit with Various Size Matrices

Since there is complete freedom as to the sizes of arrays that may be used, how is this communicated to the procedures? The obligation to accurately specify this is now completely on the programmer; the compiler is no longer any help.

Let us use **mfadd** as an example. Suppose we have the declarations

```
type
    array4x5 : array [ 1 .. 4, 1 .. 5 ] of float ;
var
    A, B, C : array4x5 ;
```

and we wish to compute the sum **C:=A+B**. This is done by the statement

```
mfadd( C, A, B, 4 * 5 ) ;
```

Note that arrays that have subscript ranges that begin at numbers other than 1 will require special care.

4. USING THE APPENDICES

4.1 How to find a procedure

- First look in the section PROCEDURE DESCRIPTIONS for the name of the procedure that satisfies the need. If there is none, a new one must be written.
- Next, examine the CALLING SEQUENCES to determine the parameters of the procedure.
- The EXAMPLES may clarify any lingering doubts.
- The last recourse is to the the code itself, in the APPENDICES.

4.2 Conventions in the code

Several mnemonic devices are employed in the **vectors** and **matrices** units:

- The first letter indicates the type of operation: the obvious *v* for vector and *m* for matrix. In the matrix operations, the second letter is also an indicator: *f* indicates a "floating point" operation, and the letter *i* indicates an "integer" operation.
- The order of parameters in the calling sequence has the same order as in an assignment statement. Thus, since assignment works from right to left, the parameter list works the same. For example, given the scalars *a, b, c* and the vectors *a1, b1, c1:* the assignment of the scalar sum has the form

  ```
  c := a + b
  ```

  The vector addition precisely parallels the assignment statement, *viz.*

  ```
  vadd( c1, a1, b1 )
  ```

  The exception to this rule occurs when files are involved; in this case, the parameter lists are patterned after the "read" and "write" statements: the file parameter comes first, followed by the entity to be "i/o-ed".

- Finally, *functions* return their result as both the function value itself, and also as the last parameter of the call. This caters for constructs of the form

  ```
  if  function_call( ..., x ) >= 0.0  then
    statement1 ;
  if  x = 0.0  then
    statement2 ;
  ```

  which saves a reevaluation of the function.

5. CONCLUDING REMARKS

The capability to write general purpose procedures in Turbo Pascal has been demonstrated. If a programmer is willing to sacrifice the protection that Pascal offers, then this capability, along with the very fast compiler Borland produces, makes Turbo Pascal a convenient environment in which to develop engineering and scientific software.

6. REFERENCES

1. Jensen, K. and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, 1974.
2. —, *Turbo Pascal User's Guide*, Version 5.0, Borland International, 1988.
3. —, *Turbo Pascal Reference Guide*, Version 5.0, Borland International, 1988.
4. Wirth, Niklaus, *MODULA-2*, Eidgenössische Technische Hochschule Zürich, Institut für Informatik, Report 36, March 1980.

APPENDIX A - PROCEDURE DESCRIPTIONS

Vector Procedure Descriptions		
proc	vadd	vector addition: **v3** := **v1**+**v2**
proc	vcopy	vector copy: **v2** := **v1**
proc	vcross	vector cross product: **v3** := **v1** $\times$ **v2**
float	vdot	vector dot product: x := **v1** $\cdot$ **v2**
boolean	vequal	returns truth of the expression **v1**=**v2**
float	vlength	vector length: x := sqrt(**v1** $\cdot$ **v1**)
proc	vscale	**v2** := c * **v1**.
proc	vset	set all components of **v** to c.
proc	vsub	vector subtraction: **v3** := **v1**- **v2**
proc	vunit	produces unit vector **v2** in direction of **v1**
proc	vwrite	print a vector on default output (= the screen)

Matrix Procedure Descriptions		
proc	miadd	integer matrix addition: **C** :=**A**+**B**
proc	micopy	integer matrix copy: **B** :=**A**
proc	mikxm	integer scalar times integer matrix: **B** :=**k*A**
proc	miprint	print **A** per 'style' ; (style < 0) = print $\mathbf{A}^{\mathbf{T}}$
proc	miread	read integer matrix from <f>
proc	miset	set all values of integer matrix **A** to scalar k
proc	misub	integer matrix subtraction: **C** :=**A**- **B**
proc	mixpos	integer matrix transposition: **B** :=$\mathbf{A}^{\mathbf{T}}$
proc	mfadd	float matrix addition: **C** :=**A**+**B**
proc	mfcopy	float matrix copy: **B** :=**A**
boolean	mfinvert	float matrix inverse, pivot method
proc	mfkxm	float scalar times float matrix: **B** :=**k*A**
proc	mfmtxm	float matrix transpose multiplication: **C** :=$\mathbf{A}^{\mathbf{T}} \times \mathbf{B}$
proc	mfmult	float matrix multiplication: **C** :=**A** $\times$ **B**
float	mfnorm	float matrix norm
proc	mfprint	print **A** per 'style' ; (style < 0) = print $\mathbf{A}^{\mathbf{T}}$
proc	mfread	read float matrix from <f>
proc	mfset	set all values of float matrix **A** to scalar k
proc	mfsub	float matrix subtraction: **C** :=**A**- **B**
proc	mfxpos	float matrix transposition: **B** :=$\mathbf{A}^{\mathbf{T}}$

APPENDIX B - VECTOR CALLING SEQUENCES

```
procedure   vadd (      VAR   v3,     { out }
                              v1,     { in }
                              v2      { in }     : vector ) ;

procedure   vcopy (     VAR   v2,     { out }
                              v1      { in }     : vector ) ;

procedure   vcross (    VAR   v3,     { out }
                              v1,     { in }
                              v2      { in }     : vector ) ;

function    vdot (      VAR   v1,     { in }
                              v2      { in }     : vector ;
                        VAR   vd      { out }    : float
            ) : float ;

function    vequal (    VAR   v1,     { in }
                              v2      { in }     : vector ;
                              delta   { in }     : float ;
                        VAR   ve      { out}     : boolean
            ) : boolean ;

function    vlength (   VAR   v       { in }     : vector ;
                        VAR   vlen    { out }    : float
            ) : float ;

procedure   vscale (    VAR   v2      { out }    : vector ;
                              c       { in }     : float ;
                              v1      { in }     : vector ) ;

procedure   vset (      VAR   v       { out }    : vector ;
                              c       { in }     : float ) ;

procedure   vsub (      VAR   v3,     { out }
                              v1,     { in }
                              v2      { in }     : vector ) ;

procedure   vunit (     VAR   v2,     { out }
                              v1      { in }     : vector ) ;

procedure   vwrite (    VAR   v       { in }     : vector ) ;
```

APPENDIX C - MATRIX CALLING SEQUENCES

```
procedure   miadd(      VAR   c,                         { out }
                              a,                         { in }
                              b ; { untyped }            { in }
                              n: integer ) ;             { in }

procedure   micopy(     VAR   b,                         { out }
                              a ; { untyped }            { in }
                              n: integer ) ;             { in }

procedure   mikxm(      VAR   b ; { untyped }            { out }
                              k: integer ;               { in }
                        VAR   a ; { untyped }            { in }
                              n: integer ) ;             { in }

procedure   miprint(    VAR   f: text ;                  { in }
                              name: string ;             { in }
                        VAR   a ; { untyped }            { in }
                              r,                         { in }
                              c: integer ;               { in }
                              style: integer ) ;         { in }

procedure   miread(     VAR   f: text ;                  { in }
                        VAR   a ; { untyped }            { out }
                              L,                         { in }
                              M: integer ) ;             { in }

procedure   miset(      VAR   a ; { untyped }            { out }
                              k: integer ;               { in }
                              n: integer ) ;             { in }

procedure   misub(      VAR   c,                         { out }
                              a,                         { in }
                              b ; { untyped }            { in }
                              n: integer ) ;             { in }

procedure   mixpos(     VAR   b,                         { out }
                              a ; { untyped }            { in }
                              L,                         { in }
                              M : integer ) ;            { in }

procedure   mfadd(      VAR   c,                         { out }
                              a,                         { in }
                              b ; { untyped }            { in }
                              n: integer ) ;             { in }

procedure   mfcopy(     VAR   b,                         { out }
                              a ; { untyped }            { in }
                              n: integer ) ;             { in }

function    mfinvert(   VAR   a ; { untyped }            { in & out }
                              n: integer ;               { in }
```

```
                        VAR   determ : float ;          { out }
                        VAR   ierr : integer            { out }
            ) : boolean ;

procedure   mfkxm(      VAR   b ; { untyped }           { out }
                              k: float ;                { in }
                              VAR a ; { untyped }       { in }
                              n: integer ) ;            { in }

procedure   mfmtxm(     VAR   c,                        { out }
                              a,                        { in }
                              b ; { untyped }           { in }
                              L,                        { in }
                              M,                        { in }
                              N: integer ) ;            { in }

procedure   mfmult(     VAR   c,                        { out }
                              a,                        { in }
                              b ; { untyped }           { in }
                              L,                        { in }
                              M,                        { in }
                              N : integer ) ;           { in }

function    mfnorm(     VAR   a ; { untyped }           { out }
                              n: integer ;              { in }
                              norm: float               { out }
            ) : float ;

procedure   mfprint(    VAR   f: text ;                 { in }
                              name: string ;            { in }
                        VAR   a ; { untyped }           { in }
                              r,                        { in }
                              c: integer ;              { in }
                              style: integer ) ;        { in }

procedure   mfread(     VAR   f: text ;                 { in }
                        VAR   a ; { untyped }           { out }
                              L,                        { in }
                              M: integer ) ;            { in }

procedure   mfset(      VAR   a ; { untyped }           { out }
                              k: float ;                { in }
                              n: integer ) ;            { in }

procedure   mfsub(      VAR   c,                        { out }
                              a,                        { in }
                              b ; { untyped }           { in }
                              n: integer ) ;            { in }

procedure   mfxpos(     VAR   b,                        { out }
                              a ; { untyped }           { in }
                              L,                        { in }
                              M : integer ) ;           { in }
```

APPENDIX D - VECTOR EXAMPLES

Assume the following declarations apply:

```
{$I float.inc }

  VAR
    v1, v2, v3, v4, v5, v6, v7, v8, v9 : vector ;
    eq : boolean ;
    d, lenv : float ;
```

An example of invocation of each of the vector procedures is given in the program fragment below:

```
v1[1] := 1.0 ;
v1[2] := 0.0 ;
v1[3] := 0.0 ;

v2[1] := 0.0 ;
v2[2] := 1.0 ;
v2[3] := 0.0 ;
                                   { v3 := v1 + v2 }
vadd( v3, v1, v2 ) ;
                                   { v4 := v3 }
vcopy( v4, v3 ) ;
                                   { v5 := v1 × v2 }
vcross ( v5, v1, v2 ) ;
                                   { v6 := v1 · v2 }
d := vdot( v1, v2, d ) ;
                                   { eq := v1 = v2 within 0.01 }
eq := vequal ( v1, v2, 0.01, eq ) ;
                                   { lenv := √(v1· v1) }
lenv := vlength ( v1, lenv ) ;
                                   { v7 := lenv * v2 }
vscale ( v7, lenv, v2 ) ;
                                   { v8 := [ 0, 0, 0 ] }
vset ( v8, 0.0 ) ;
                                   { v9 := v8 - v7 }
vsub ( v9, v8, v7 ) ;
                                   { v9 := |v9 | }
vunit ( v9, v9 ) ;
                                   { print v9 to the screen }
vwrite ( v9 ) ;
```

APPENDIX E - MATRIX EXAMPLES

11.1 Integer Examples

Assume the following declarations apply:

```
TYPE
  iarray8x4 : array [ 1 .. 8, 1 .. 4 ] of integer ;
  iarray4x8 : array [ 1 .. 4, 1 .. 8 ] of integer ;
VAR
  ia8x4, ib8x4, ic8x4 : iarray8x4 ;
  ig4x8 : iarray4x8 ;

  f, g : text ;
```

An example of invocation of each of the integer matrix procedures is given in the program fragment below:

```
                                   { ia8x4 := [ 1 ] }
miset( ia8x4, 1, 8 * 4 ) ;
                                   { ib8x4 := [ 2 ] }
miset( ib8x4, 2, 8 * 4 ) ;
                                   { ic8x4 := ia8x4 + ib8x4 }
miadd( ic8x4, ia8x4, ib8x4, 8 * 4 ) ;
                                   { ib8x4 := ia8x4 }
micopy( ib8x4, ia8x4, 8 * 4 ) ;
                                   { ib8x4 := 3 * ib8x4 }
mikxm( ib8x4, 3, ib8x4, 8 * 4 ) ;
                                   { print ib8x4 to <f> }
miprint( f, 'ib8x4', ib8x4, 8, 4, 1 )
                                   { read ic8x4 from <g> }
miread( g, ic8x4, 8, 4 ) ;
                                   { ic8x4 := ia8x4 - ib8x4 }
misub( ic8x4, ia8x4, ib8x4, 8 * 4 ) ;
                                   { ig4x8 := ic8x4^T }
mixpos( ig4x8, ia8x4, 8, 4 ) ;
```

11.2 Float Examples

Assume the following declarations apply:

```
{$I float.inc }
  TYPE
    array8x4 : array [ 1 .. 8, 1 .. 4 ] of float ;
    array4x4 : array [ 1 .. 4, 1 .. 4 ] of float ;
    array4x8 : array [ 1 .. 4, 1 .. 8 ] of float ;
  VAR
    a8x4, b8x4, c8x4 : array8x4 ;
    d4x4, e4x4, f4x4 : array4x4 ;
    g4x8, h4x8, i4x8 : array4x8 ;
    determ : float ;
    inverse_error : integer ;
    f, g : text ;
    e4x4_norm : float ;
```

An example of invocation of each of the float matrix procedures is given on the next page.

```
                                    { a8x4 := [ 1.0 ] }
mfset( a8x4, 1.0, 8 * 4 ) ;
                                    { b8x4 := [ 2.0 ] }
mfset( b8x4, 2.0, 8 * 4 ) ;
                                    { d4x4 := [ 3.0 ] }
mfset( d4x4, 3.0, 4 * 4 ) ;
                                    { g4x8 := [ 4.0 ] }
mfset( g4x8, 4.0, 4 * 8 ) ;

                                    { c8x4 := a8x4 + b8x4 }
mfadd( c8x4, a8x4, b8x4, 8 * 4 ) ;
                                    { b8x4 := a8x4 }
mfcopy( b8x4, a8x4, 8 * 4 ) ;
                                    { a8x4 := c8x4 × d4x4 }
mfmult( a8x4, c8x4, d4x4, 8, 4, 4 ) ;
                                    { e4x4 := g4x8 × a8x4 }
mfmult( e4x4, g4x8, a8x4, 4, 8, 4 ) ;
                                    { try to invert e4x4 }
if  mfinvert( e4x4, 4, determ, inverse_error ) then
  mfprint( f, 'E4x4 Inverse', e4x4, 4, 4, 2 )
else begin
  if  inverse_error = 1  then begin
    writeln( 'Out of memory trying to invert E4x4.' ) ;
    halt( 1 )
  end
  else if  inverse_error = 2  then begin
    writeln( 'Inverse doesn''t exist.' ) ;
    halt( 1 )
  end
  else begin
    writeln( 'Can''t happen!' ) ;
    halt( 1 )
  end
end ;
                                    { b8x4 := π * a8x4 }
mfkxm( b8x4, PI, a8x4, 8 * 4 ) ;
                                    { e4x4 := b8x4^T × b8x4 }
mfmtxm( e4x4, b8x4, b8x4, 4, 8, 4 ) ;
                                    { find a norm of e4x4 }
e4x4_norm := mfnorm( e4x4, 4 * 4, e4x4_norm ) ;
                                    { read h4x8 from <g> }
mfread( g, h4x8, 4, 8 ) ;
                                    { c8x4 := a8x4 - b8x4 }
mfsub( c8x4, a8x4, b8x4, 8 * 4 ) ;
                                    { g4x8 := b8x4^T }
mfxpos( g4x8, b8x4, 8, 4 ) ;
```

APPENDIX F - THE VECTORS UNIT SOURCE CODE

```
Unit vectors ;

(*. vectors - in 3-space. *)

Interface              (* public declarations *)

{$I float.inc }                  (* defines numerical precision *)

TYPE
  vector = array [ 1 .. 3 ] of float ;

procedure  vadd    ( VAR v3, v1, v2 : vector ) ;
procedure  vcopy   ( VAR v2, v1     : vector ) ;
procedure  vcross  ( VAR v3, v1, v2 : vector ) ;
function   vdot    ( VAR v1, v2     : vector ; VAR vd : float )
                   : float ;
function   vequal  ( VAR v1, v2     : vector ; delta: float ; ve : boolean )
                   : boolean ;
function   vlength ( VAR v          : vector ; VAR vlen : float )
                   : float ;
procedure  vscale  ( VAR v2         : vector ; c : float ; v1 : vector ) ;
procedure  vset    ( VAR v          : vector ; c : float ) ;
procedure  vsub    ( VAR v3, v1, v2 : vector ) ;
procedure  vunit   ( VAR v2, v1     : vector ) ;
procedure  vwrite  ( VAR v          : vector ) ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                   (* private declarations *)

(*****
 *
 *.   vadd - vector addition:  {v3} := {v1} + {v2}
 *
 *)
procedure  vadd    ( VAR v3, v1, v2 : vector ) ;
  VAR i : 1 .. 3 ;
  begin
    for  i := 1  to  3  do
      v3[i] := v1[i] + v2[i]
  end (* vadd *) ;

(*****
 *
 *.   vcopy - vector copy:  {v2} := {v1}
 *
 *)
procedure  vcopy   ( VAR v2, v1     : vector ) ;
  VAR i : 1 .. 3 ;
  begin
    for  i := 1  to  3  do
      v2[i] := v1[i]
  end (* vcopy *) ;

(*****
 *
 *.   vcross - vector cross product:  {v3} := {v1} X {v2}
 *
 *)
procedure  vcross  ( VAR v3, v1, v2 : vector ) ;
  begin
    v3[1] := v1[2] * v2[3] - v1[3] * v2[2] ;
    v3[2] := v1[3] * v2[1] - v1[1] * v2[3] ;
```

```
    v3[3] := v1[1] * v2[2] - v1[2] * v2[1]
  end (* vcross *) ;

(*****
 *
 *.   vdot - vector dot product: x := {v1} * {v2}
 *
 *  Note that the dot product is returned both as function value
 *  and as the 3rd argument.  This can be useful in constructs
 *  of the form
 *      if  vdot( a, b, vd ) > 0  then
 *            if  vd < 3.0  then  etc.
 *
 *)
function    vdot    ( VAR v1, v2     : vector ; VAR vd : float )
                    : float ;
  VAR i : 1 .. 3 ;
      z : float ;
  begin
    z := 0.0 ;
    for  i := 1  to  3  do
      z := z + v1[i] * v2[i] ;
    vd   := z ;
    vdot := z
  end (* vdot *) ;

(*****
 *
 *.   vequal - returns truth of the expression {v1} = {v2},
 *             within the tolerance DELTA (3rd argument).
 *
 *  Note that the result is returned both as function value
 *  and as the 4th argument.  This can be useful in constructs
 *  of the form
 *
 *     if  vequal( a, b, delta, vd )  then
 *           do something ;
 *     ...
 *     if  vd  then     <- remembering the result of the test
 *        do something else ;
 *
 *)
function    vequal  ( VAR v1, v2     : vector ; delta: float ; ve : boolean )
                    : boolean ;
  VAR
    i : 1 .. 3 ;
    status : ( equal, notequal, unknown ) ;
  begin

    status := unknown ;   i := 1 ;
    while  status = unknown  do
      if  v1[i] - v2[i] > delta  then
        status := notequal
      else if  i = 3  then
        status := equal
      else
        i := i + 1 ;

    ve     := status = equal ;
    vequal := ve
  end (* vequal *) ;
```

```
(*****
 *
 *.   vlength - vector length: x := sqrt( {v1} * {v1} )
 *
 *  Note that length is returned both as function value
 *  and as the 2nd argument.  This can be useful in constructs
 *  of the form
 *      if  vlength( a, x ) > 0  then
 *            if  x < 3.0  then  etc.
 *
 *)
function   vlength ( VAR v          : vector ; VAR vlen : float )
                   : float ;
  VAR i : 1 .. 3 ;
      z : float ;
  begin

    z := 0.0 ;
    for  i := 1  to  3  do
      z := z + sqr ( v[i] ) ;

    vlen    := sqrt ( z ) ;
    vlength := vlen
  end (* vlength *) ;

(*****
 *
 *.  vscale - {v2} := c * {v1}.
 *
 *)
procedure  vscale  ( VAR v2        : vector ; c : float ; v1 : vector ) ;
  VAR i : 1 .. 3 ;
  begin
    for  i := 1  to  3  do
      v2[i] := c * v1[i]
  end (* vscale *) ;

(*****
 *
 *.  vset - set all components of {v} to c.
 *
 *)
procedure  vset    ( VAR v         : vector ; c : float ) ;
  VAR i : 1 .. 3 ;
  begin
    for  i := 1  to  3  do
      v[i] := c
  end (* set *) ;

(*****
 *
 *.   vsub - vector subtraction:  {v3} := {v1} - {v2}
 *
 *)
procedure  vsub    ( VAR v3, v1, v2 : vector ) ;
  VAR i : 1 .. 3 ;
  begin
    for  i := 1  to  3  do
      v3[i] := v1[i] - v2[i]
  end (* vsub *) ;

(*****
 *
```

```
 *.    unit - produces unit vector {v2} in direction of {v1}
 *
 *  Note: if the length of {v1} is 0, then {v2} = {0}.
 *
 *)
procedure  vunit   ( VAR v2, v1     : vector ) ;
  VAR i : 1 .. 3 ;
      z : float ;
  begin
    if  vlength ( v1 , z ) > 0.0  then
      vscale( v2, 1.0 / z, v1 )
    else
      vcopy ( v2, v1 )
  end (* vunit *) ;

(*****
 *
 *.  vwrite - print a vector on default output (= the screen).
 *
 *)
procedure  vwrite  ( VAR v          : vector ) ;
  VAR i : 1 .. 3 ;
  begin
    for  i := 1  to  3  do
      write ( v[i] ) ;
    writeln
  end (* vwrite *) ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)

end.
```

```
Unit matrices ;

(*. matrices  -  provides variable dimension matrix math. *)

Interface             (* public declarations *)

{$I float.inc }                  (* defines numerical precision *)

procedure   miadd    ( VAR c, a, b ;   n: integer ) ;
procedure   micopy   ( VAR b, a    ;   n: integer ) ;
procedure   mikxm    ( VAR b       ;   k: integer ;   VAR a ;   n: integer ) ;
procedure   miprint  ( VAR f: text ;   name: string ;   VAR a ;
                       r, c: integer ;   style: integer ) ;
procedure   miread   ( VAR f: text ;   VAR a ;   L, M: integer ) ;
procedure   miset    ( VAR a       ;   k: integer ;   n: integer ) ;
procedure   misub    ( VAR c, a, b ;   n: integer ) ;
procedure   mixpos   ( VAR b, a    ;   L, M : integer ) ;
procedure   mfadd    ( VAR c, a, b ;   n: integer ) ;
procedure   mfcopy   ( VAR b, a    ;   n: integer ) ;
function    mfinvert( VAR a       ;   n: integer ;   VAR determ : float ;
                       VAR ierr : integer ) : boolean ;
procedure   mfkxm    ( VAR b       ;   k: float ;   VAR a ;   n: integer ) ;
procedure   mfmtxm   ( VAR c, a, b ;   L, M, N: integer ) ;
procedure   mfmult   ( VAR c, a, b ;   L, M, N : integer ) ;
function    mfnorm   ( VAR a       ;   n: integer ;   norm: float ) : float ;
procedure   mfprint  ( VAR f: text ;   name: string ;   VAR a ;
                       r, c: integer ;   style: integer ) ;
procedure   mfread   ( VAR f: text ;   VAR a ;   L, M: integer ) ;
procedure   mfset    ( VAR a       ;   k: float ;   n: integer ) ;
procedure   mfsub    ( VAR c, a, b ;   n: integer ) ;
procedure   mfxpos   ( VAR b, a    ;   L, M : integer ) ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                   (* private declarations *)

const
  MAX_FLOAT_MATRIX = 65520 div Sizeof( float ) ;
  MAX_FLOAT_STYLES = 2 ;              (* for printing *)

  MAX_INT_MATRIX = 65520 div Sizeof ( integer ) ;
  MAX_INT_STYLES = 2 ;              (* for printing *)

type
  float_matrix   = array [ 1 .. MAX_FLOAT_MATRIX ] of float ;
  integer_matrix = array [ 1 .. MAX_INT_MATRIX   ] of integer ;

VAR
                                    (* for mfprint *)
  float_colwidth_per_style : array [ 1 .. MAX_FLOAT_STYLES ] of integer ;
  float_ncols_per_style    : array [ 1 .. MAX_FLOAT_STYLES ] of integer ;
  float_sigfigs_per_style  : array [ 1 .. MAX_FLOAT_STYLES ] of integer ;

                                    (* for miprint *)
  int_colwidth_per_style : array [ 1 .. MAX_INT_STYLES ] of integer ;
  int_ncols_per_style    : array [ 1 .. MAX_INT_STYLES ] of integer ;

(*****
 *
 *     A few internal utility routines,
 *     to make the unit self contained.
 *
 *)
```

```
      (*****
       *
       *.   imin - return smaller integer of a, b.
       *
       *)
      function imin ( a, b: integer ) : integer ;
        begin
          if a <= b  then  imin := a  else  imin := b
        end (* imin *) ;

      (*****
       *
       *.   imax - return larger integer of a, b.
       *
       *)
      function imax ( a, b: integer ) : integer ;
        begin
          if a >= b  then  imax := a  else  imax := b
        end (* imax *) ;

      (*****
       *
       *.   fmax - return larger float of a, b.
       *
       *)
      function fmax ( a, b: float ) : float ;
        begin
          if a >= b  then  fmax := a  else  fmax := b
        end (* fmax *) ;

     (*****
      *
      *.   lss - linear sub_script for [i, j] into [1..?, 1..N] matrix.
      *
      *)
     function lss ( i, j, N: integer ) : integer ;
        begin
              lss := ( i - 1 ) * N + j
        end (* lss *) ;

  (*
   *
   *     End internal routines
   *
   ******)

  (*****
   *
   *.   miadd - integer matrix addition: [c] := [a] + [b].
   *
   *)
  procedure miadd ( VAR c, a, b ;   n: integer ) ;

    VAR
      cc : integer_matrix  absolute  c ;
      aa : integer_matrix  absolute  a ;
      bb : integer_matrix  absolute  b ;

    begin (* miadd *)

      for  n := 1  to  n  do
         cc[n] := aa[n] + bb[n]
```

```
  end (* miadd *) ;

 (*****
  *
  *.   micopy - integer matrix copy: [b] := [a].
  *
  *)
procedure micopy ( VAR b, a ;   n: integer ) ;

  VAR
    aa : integer_matrix  absolute  a ;
    bb : integer_matrix  absolute  b ;

  begin (* micopy *)

    for  n := 1  to  n  do
       bb[n] := aa[n]

  end (* micopy *) ;

 (*****
  *
  *.   mikxm - integer scalar * integer matrix: [b] := k*[a].
  *
  *)
procedure mikxm ( VAR b ;   k: integer ;   VAR a ;   n: integer ) ;

  VAR
    bb : integer_matrix  absolute  b ;
    aa : integer_matrix  absolute  a ;

  begin (* mikxm *)

    for  n := 1  to  n  do
       bb[n] := k * aa[n]

  end (* mikxm *) ;

 (*****
  *
  *.    miprint - print [a] per 'style' ; (style < 0) = 'print [a] transpose.'
  *
  *)
procedure miprint ( VAR f: text ;     name: string ;     VAR a ;
                     r, c: integer ;   style: integer ) ;

VAR
  aa :          integer_matrix  absolute  a ;
  cbeg :        integer ;
  cend  :       integer ;
  colwidth :    integer ;
  ic :          integer ;
  j :           integer ;
  jk :          integer ;
  k :           integer ;
  kj :          integer ;
  maxsty :      integer ,
  nc :          integer ;
  ns :          integer ;
  styabs :      integer ;

  (*****
   *
```

```
 *.   miprint_name - output the array name string.
 *
 *)
procedure miprint_name ;
  begin
    writeln( f ) ;
    writeln( f, ' ... ', name, ' ...' ) ;
    writeln( f )
  end (* miprint_name *) ;

(*****
 *
 *.   miprint_headings - print the column headings.
 *
 *)
procedure miprint_headings ;
  VAR
    j, k : integer ;
  begin

                                  (* write the column headings *)
     write( f, ' ': 5 ) ;                       (* tab over 5 *)
     write( f, cbeg : colwidth div 2 + 1 ) ; (* 1st heading *)
     for  j := cbeg + 1  to  cend  do         (* rest of headings *)
        write( f, j : colwidth ) ;
     writeln( f ) ;

                                  (* underline the column headings *)
     write( f, ' ': 5 ) ;                       (* tab over 5 *)
     for  j := cbeg to  cend  do begin        (* all of the headings *)
        write( f, ' ':1 ) ;
        for  k := 2  to  colwidth  do
           write( f, '-' : 1 ) ;
     end ;
     writeln( f )

  end (* miprint_headings *) ;

begin (* --- miprint --- *)

   styabs   := abs( style ) ;  (* positive style number *)
   ns       := imax( 1, imin( styabs, MAX_INT_STYLES ) ) ;  (* insurance *)
   nc       := int_ncols_per_style[ ns ] ;
   colwidth := int_colwidth_per_style[ ns ] ;
   cend     := 0 ;

   miprint_name ;
                                    (* print normally, not the transpose *)
   if  style > 0  then begin
      ic := 1 ;
      while  cend < c  do begin
         cbeg := imin( ic, c ) ;
         cend := imin( ic + nc - 1, c ) ;
         if  ic > 1  then
            writeln( f ) ;

         miprint_headings ;

                                    (* write the array values *)
         for  k := 1  to  r  do begin
            write( f, k : 5 ) ;
               kj := lss( k, cbeg, c ) ;
```

```
          for  j := cbeg  to  cend  do begin
                write( f, aa[kj] : colwidth ) ;
                inc( kj )
          end ;
          writeln( f )
        end ;

                                        (* bump column index *)
        ic := ic + nc
      end (* while *)
    end (* if *)

                                        (* print transpose *)
    else begin
      ic := 1 ;
      while  cend < r  do begin
        cbeg := imin( ic, r ) ;
        cend := imin( ic + nc - 1, r ) ;
        if  ic > 1  then
          writeln( f ) ;

        miprint_headings ;

        for  k := 1  to  c  do begin
          write( f, k : 5 ) ;
             jk := lss( cbeg, k, c ) ;
          for  j := cbeg  to  cend  do begin
                write( f, aa[jk] : colwidth ) ;
                inc( jk, c )
          end ;
          writeln( f )
        end ;

        ic := ic + nc

      end (* while *)
    end (* else *) ;

    flush( f )

  end (* miprint *) ;

  (*****
   *
   *.   miread - read integer matrix from <f>.
   *
   *)
  procedure miread ( VAR f: text ;   VAR a ;   L, M: integer ) ;

    VAR
      aa : integer_matrix  absolute  a ;
      i, j, ij : integer ;

    begin (* miread *)

      for  i := 1  to  L  do begin
            ij := lss( i, 1, M ) ;
        for  j :=  1  to  M  do begin
              read( f, aa[ij] ) ;
              inc( ij )
        end ;
        readln( f )
      end
```

```
  end (* miread *) ;

(*****
 *
 *.   miset - set all values of integer matrix [a] to scalar  k .
 *
 *)
procedure miset ( VAR a ;   k: integer ;   n: integer ) ;

  VAR
    aa : integer_matrix  absolute  a ;

  begin (* miset *)

    for  n := 1  to  n  do
       aa[n] := k

  end (* miset *) ;

(*****
 *
 *.   misub - integer matrix subtraction: [c] := [a] - [b].
 *
 *)
procedure misub ( VAR c, a, b ;   n: integer ) ;

  VAR
    cc : integer_matrix  absolute  c ;
    aa : integer_matrix  absolute  a ;
    bb : integer_matrix  absolute  b ;

  begin (* misub *)

    for  n := 1  to  n  do
       cc[n] := aa[n] - bb[n]

  end (* misub *) ;

(*****
 *
 *.   mixpos - integer matrix transposition: [b] := [a]T.
 *
 *)
procedure mixpos ( VAR b, a ;   L, M : integer ) ;

  VAR
    aa: integer_matrix  absolute  a ;
    bb: integer_matrix  absolute  b ;
    i, j: integer ;
    ij, ji: integer ;

  begin (* mixpos *)

     for  i := 1  to  L  do begin
           ij     := lss( i, 1, M ) ;
           ji     := lss( 1 ,i, L ) ;
        for  j := 1  to  M  do begin
              bb[ji] := aa[ij] ;
              inc( ij ) ;
              inc( ji, L )
           end
     end
```

```
  end (* mixpos *) ;

(*****
 *
 *.   mfadd - float matrix addition: [c] := [a] + [b].
 *
 *)
procedure mfadd ( VAR c, a, b ;   n: integer ) ;

  VAR
    cc : float_matrix  absolute  c ;
    aa : float_matrix  absolute  a ;
    bb : float_matrix  absolute  b ;

  begin (* mfadd *)

    for  n := 1  to  n  do
       cc[n] := aa[n] + bb[n]

  end (* mfadd *) ;

(*****
 *
 *.   mfcopy - float matrix copy: [b] := [a].
 *
 *)
procedure mfcopy ( VAR b, a ;   n: integer ) ;

  VAR
    aa : float_matrix  absolute  a ;
    bb : float_matrix  absolute  b ;

  begin (* mfcopy *)

    for  n := 1  to  n  do
       bb[n] := aa[n]

  end (* mfcopy *) ;

(*****
 *
 *.   mfinvert - float matrix inverse, pivot method:
 *      The function value is returned TRUE for success, FALSE otherwise
 *      a[1..n,1..n] is returned as its own inverse.
 *      determ       is returned as the determinant
 *      ierr =  0 means success ( same as TRUE for function value )
 *              1 means 'OUT OF MEMORY'
 *              2 no inverse exists
 *
 *     Adapted from original routine  an f402 (share)
 *     by S Good, November  1971, at the David Taylor Model Basin
 *     Test for loss of digits due to subtraction   C.R. Newman, NOL 1/70
 *
 *)
function mfinvert   ( VAR a ;   n : integer ;   VAR determ: float ;
                      VAR ierr : integer ) : boolean ;
label 13 ;

type
  introw_array = array [ 1 .. MAX_FLOAT_MATRIX ] of integer ;
  introw = ^introw_array ;

VAR
  aa :           float_matrix absolute a ;
```

```
        bmax :          float ;
        eps :           float ;
        i :             integer ;
        icolum :        integer ;

        ind1 :          introw ;
        ind2 :          introw ;
        ind3 :          introw ;

        invert :        boolean ;
        irow :          integer ;
        j :             integer ;
        k :             integer ;
        p :             integer ;
        pivot :         float ;
        q :             integer ;
        r :             integer ;
        s :             integer ;
        t :             integer ;
        sndet :         float ;
        sub :           float ;
        temp :          float ;

    begin
       (*****
        *
        *      initialization
        *
        *)
       eps    := 1.0e-14 ;
       invert := true ;
       ierr   := 0 ;
       determ := 1.0 ;
       sndet  := 1.0 ;

       if memavail < 3 * n * sizeof( integer ) then begin
          invert := false ;
          ierr   := 1 ;
          goto 13
       end
       else begin
          getmem( ind1, n * sizeof( integer ) ) ;
          getmem( ind2, n * sizeof( integer ) ) ;
          getmem( ind3, n * sizeof( integer ) )
       end (* else *) ;

       miset( ind3^, 0, n ) ;

       (*****
        *
        *     search for pivot elements
        *
        *)
       for  i := 1  to  n  do begin
          temp := 0.0 ;
          for  j := 1  to  n  do
             if  ind3^[j] <> 1  then
                for  k := 1  to  n  do
                   if  ind3^[k] < 1  then begin
                      r := lss( j, k, n ) ;
                      if  temp < abs( aa[r] )  then begin
                         irow   := j ;
```

```
                    icolum := k ;
                    temp   := abs( aa[r] )
                 end
              end
              else if  ind3^[k] <> 1  then begin
                  invert := false ;
                  ierr   := 2 ;
                  goto 13
              end ;
     ind3^[icolum] := ind3^[icolum] + 1 ;
     ind1^[i] := irow ;
     ind2^[i] := icolum ;
     if  temp = 0.0  then begin
        invert := false ;
        ierr   := 2 ;
        goto 13
     end ;

  (*****
   *
   *     interchange rows to put pivot element on diagonal
   *
   *)
     if  irow <> icolum  then begin
        sndet := -sndet ;
           r     := lss( irow, 1, n ) ;
           s     := lss( icolum, 1, n ) ;
        for  p := 1  to  n  do begin
           temp  := aa[r] ;
           aa[r] := aa[s] ;
              aa[s] := temp ;
              inc( r ) ;
              inc( s )
        end
     end ;

  (*****
   *
   *     divide pivot row by pivot element
   *
   *)
     r      := lss( icolum, icolum, n ) ;
     pivot  := aa[r] ;
     if  pivot <> 0.0  then
           temp := 1.0 / pivot
     else
           temp := 0.0 ;
     determ := determ * pivot ;
     aa[r]  := 1.0 ;
     r      := lss( icolum, 1, n ) ;
     for  p := 1  to  n  do begin
           aa[r] := aa[r] * temp ;
           inc( r )
     end (* for *) ;

  (*****
   *
   *     reduce non-pivot rows
   *
   *)
     bmax := 0.0 ;
     for  q := 1  to  n  do
```

```
        if  q <> icolum  then begin
           r     := lss( q, icolum, n ) ;
           temp  := aa[r] ;
           aa[r] := 0.0 ;
              s     := lss( icolum, 1, n ) ;
              t     := lss( q, 1, n ) ;
           for  p := 1  to  n  do begin
              sub   := aa[s] * temp ;
              aa[t] := aa[t] - sub ;
              if  ind3^[q] <> 1  then
                 if  abs( aa[t] ) <= eps * abs( sub )  then
                          bmax := fmax( bmax, abs( aa[t] ) ) ;
                 inc( s ) ;
                 inc( t )
           end
        end
   end (* for i := 1  to  n  do begin *) ;

   (*****
    *
    *    interchange columns
    *
    *)
   for  i := 1  to  n  do begin
      p := n + 1 - i ;
      if  ind1^[p] <> ind2^[p]  then begin
            irow   := ind1^[p] ;
         icolum := ind2^[p] ;
            r      := lss( k, irow, n ) ;
            s      := lss( k, icolum, n ) ;
         for  k := 1  to  n  do begin
            temp  := aa[r] ;
            aa[r] := aa[s] ;
               aa[s] := temp ;
               inc( r, n ) ;
               inc( s, n )
         end (* for *)
      end (* if *)
   end ;

   for  k := 1  to  n  do
      if  ind3^[k] <> 1  then begin
         invert := false ;
         ierr   := 2 ;
         goto 13
      end ;

   determ := determ * sndet ;
                                            (* return memory to the heap *)
   freemem( ind3, n * sizeof( integer ) ) ;
   freemem( ind2, n * sizeof( integer ) ) ;
   freemem( ind1, n * sizeof( integer ) ) ;

13: mfinvert := invert

end (* mfinvert *) ;

(*****
 *
 *.   mfkxm - float scalar * float matrix: [b] := k*[a].
 *
 *)
procedure mfkxm ( VAR b ;   k: float ;   VAR a ;   n: integer ) ;
```

```
  VAR
    bb : float_matrix  absolute  b ;
    aa : float_matrix  absolute  a ;

  begin (* mfkxm *)

    for  n := 1  to  n  do
       bb[n]  := k * aa[n]

  end (* mfkxm *) ;

(*****
 *
 *.   mfmtxm - float matrix transpose multiplication: [c] := [a]T * [b].
 *
 *  Assume the following dimensions:
 *    a[M,L] - note the 1st dimension matches 1st of b
 *    b[M,N]
 *    c[L,N]
 *
 *)
procedure mfmtxm ( VAR c, a, b ;   L, M, N: integer ) ;

  VAR
    cc : float_matrix  absolute  c ;
    aa : float_matrix  absolute  a ;
    bb : float_matrix  absolute  b ;
    i, j, k : integer ;
    ij, ki, kj : integer ;
    s : float ;

  begin (* mfmtxm *)

     for  i := 1  to  L  do
        for  j := 1  to  N  do begin
           s := 0.0 ;
              ki := lss( 1, i, L ) ;
              kj := lss( 1, j, N ) ;
           for  k := 1  to  M  do begin
                s  := s + aa[ki] * bb[kj] ;
                inc( ki, L ) ;
                inc( kj, N )
           end ;
              ij     := lss( i, j, N ) ;
           cc[ij] := s
        end

  end (* mfmtxm *) ;

(*****
 *
 *.   mfmult - float matrix multiplication: [c](LxN) := [a](LxM) * [b](MxN).
 *
 *
 *    Assuming the dimensions:  a[L,M],  b[M,N]  and  c[L,N],
 *    a special purpose matrix multiply might look like:
 *
 *    procedure mmult_eg ( VAR c mattypec ; VAR a mattypea ; VAR b mattypeb ) ;
 *      VAR
 *        s: float ;
 *        i,j,k: integer ;
 *      begin
```

```
 *        for  i := 1  to  L  do
 *          for  j := 1  to  N  do
 *            begin
 *              s := 0.0 ;
 *              for  k := 1  to  M  do
 *                s := s + a[i,k] * b[k,j] ;
 *              c[i,j] := s
 *            end
 *      end ;
 *
 *)
procedure mfmult ( VAR c, a, b ;   L, M, N : integer ) ;

  VAR
    cc: float_matrix  absolute  c ;
    aa: float_matrix  absolute  a ;
    bb: float_matrix  absolute  b ;
    s: float ;
    i,j,k: word ;
    ik, kj, ij: word ;

  begin (* mfmult *)

     for  i := 1  to  L  do
       for  j := 1  to  N  do begin
          s := 0.0 ;
             ik := lss( i, 1, M ) ;
             kj := lss( 1, j, N ) ;
          for  k := 1  to  M  do begin
                s  := s + aa[ik] * bb[kj] ;
                inc( ik ) ;
                inc( kj, N )
          end ;
             ij := lss( i, j, N ) ;
          cc[ij] := s ;
       end

  end (* mfmult *) ;

(*****
 *
 *.   mfnorm - float matrix norm.
 *
 *)
function mfnorm ( VAR a ;   n: integer ;   norm: float   ) : float ;

  VAR
    aa : float_matrix  absolute  a ;

  begin (* mfnorm *)

    norm := 0.0 ;
    for  n := 1  to  n  do
       norm := norm + sqr( aa[n] ) ;

    norm   := sqrt( norm ) ;
    mfnorm := norm

  end (* mfnorm *) ;

(*****
 *
```

```
 *.   mfprint - print [a] per 'style' ; (style < 0) = 'print [a] transpose.'
 *
 *)
procedure mfprint ( VAR f: text ;    name: string ;    VAR a ;
                    r, c: integer ;   style: integer ) ;

VAR
  aa :          float_matrix  absolute  a ;
  cbeg :        integer ;
  cend  :       integer ;
  colwidth :    integer ;
  ic :          integer ;
  j :           integer ;
  jk :          integer ;
  k :           integer ;
  kj :          integer ;
  maxsty :      integer ;
  nc :          integer ;
  ns :          integer ;
  sigfigs :     integer ;
  styabs :      integer ;

   (*****
    *
    *.   mfprint_name - output the array name string.
    *
    *)
    procedure mfprint_name ;
      begin
        writeln( f ) ;
        writeln( f, ' ... ', name, ' ...' ) ;
        writeln( f )
      end (* mfprint_name *) ;

   (*****
    *
    *.   mfprint_headings - print the column headings.
    *
    *)
    procedure mfprint_headings ;
      VAR
        j, k : integer ;
      begin

                                     (* write the column headings *)
         write( f, ' ': 5 ) ;                              (* tab over 5 *)
         write( f, cbeg : colwidth div 2 + 1 ) ;           (* 1st heading *)
         for  j := cbeg + 1  to  cend  do                  (* rest of headings *)
            write( f, j : colwidth ) ;
         writeln( f ) ;

                                     (* underline the column headings *)
         write( f, ' ': 5 ) ;                      (* tab over 5 *)
         for  j := cbeg to  cend  do begin         (* all of the headings *)
            write( f, ' ':1 ) ;
            for  k := 2  to  colwidth  do
               write( f, '-' : 1 ) ;
         end ;
         writeln( f )

      end (* mfprint_headings *) ;
```

```
begin (* --- mfprint --- *)

  styabs   := abs( style ) ; (* positive style number *)
  ns       := imax( 1, imin( styabs, MAX_FLOAT_STYLES ) ) ;  (* insurance *)
  nc       := float_ncols_per_style[ ns ] ;
  colwidth := float_colwidth_per_style[ ns ] ;
  sigfigs  := float_sigfigs_per_style[ ns ] ;
  cend     := 0 ;

  mfprint_name ;
                                    (* print normally, not the transpose *)
   if  style > 0  then begin
      ic := 1 ;
      while  cend < c  do begin
         cbeg := imin( ic, c ) ;
         cend := imin( ic + nc - 1, c ) ;
         if  ic > 1  then
            writeln( f ) ;

         mfprint_headings ;

                                    (* write the array values *)
         for  k := 1  to  r  do begin
            write( f, k : 5 ) ;
               kj := lss( k, cbeg, c ) ;
            for  j := cbeg  to  cend  do begin
                  write( f, aa[kj] : colwidth : sigfigs ) ;
                  inc( kj )
            end ;
            writeln( f )
         end ;

                                    (* bump column index *)
         ic := ic + nc
       end (* while *)
   end (* if *)

                                    (* print transpose *)
   else begin
      ic := 1 ;
      while  cend < r  do begin
         cbeg := imin( ic, r ) ;
         cend := imin( ic + nc - 1, r ) ;
         if  ic > 1  then
            writeln( f ) ;

         mfprint_headings ;

         for  k := 1  to  c  do begin
            write( f, k : 5 ) ;
               jk := lss( cbeg, k, c ) ;
            for  j := cbeg  to  cend  do begin
                  write( f, aa[jk] : colwidth : sigfigs ) ;
                  inc( jk, c )
            end ;
            writeln( f )
         end ;

         ic := ic + nc

      end (* while *)
   end (* else *) ;
```

```
  flush( f )

  end (* mfprint *) ;

(*****
 *
 *.   mfread - read float matrix from <f>.
 *
 *)
procedure mfread ( VAR f: text ;   VAR a ;   L, M: integer ) ;

  VAR
    aa : float_matrix  absolute  a ;
    i, j, ij : integer ;

  begin (* mfread *)

     for  i := 1  to  L  do begin
           ij := lss( i, 1, M ) ;
        for  j :=  1  to  M  do begin
              read( f, aa[ij] ) ;
              inc( ij )
        end ;
        readln( f )
     end

  end (* mfread *) ;

(*****
 *
 *.   mfset - set all values of float matrix [a] to scalar  k .
 *
 *)
procedure mfset ( VAR a ;   k: float ;   n: integer ) ;

  VAR
    aa : float_matrix  absolute  a ;

  begin (* mfset *)

    for  n := 1  to  n  do
       aa[n] := k

  end (* mfset *) ;

(*****
 *
 *.   mfsub - float matrix subtraction: [c] := [a] - [b].
 *
 *)
procedure mfsub ( VAR c, a, b ;   n: integer ) ;

  VAR
    cc : float_matrix  absolute  c ;
    aa : float_matrix  absolute  a ;
    bb : float_matrix  absolute  b ;

  begin (* mfsub *)

    for  n := 1  to  n  do
       cc[n] := aa[n] - bb[n]

  end (* mfsub *) ;
```

```
(*****
 *
 *.   mfxpos - float matrix transposition: [b] := [a]T.
 *
 *)
procedure mfxpos ( VAR b, a ;   L, M : integer ) ;

  VAR
    aa: float_matrix  absolute  a ;
    bb: float_matrix  absolute  b ;
    i, j: integer ;
    ij, ji: integer ;

  begin (* mfxpos *)

     for  i := 1  to  L  do begin
            ij := lss( i, 1, M ) ;
            ji := lss( 1 ,i, L ) ;
        for  j := 1  to  M  do begin
            bb[ji] := aa[ij] ;
            inc( ij ) ;
            inc( ji, L )
            end
     end

  end (* mfxpos *) ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
begin (* initialization code *)

                            (* for mfprint *)
  float_ncols_per_style    [ 1 ] :=   4 ;
  float_colwidth_per_style[ 1 ] :=  17 ;
  float_sigfigs_per_style [ 1 ] := - 8 ;      (* forces E format *)

  float_ncols_per_style    [ 2 ] :=   8 ;
  float_colwidth_per_style[ 2 ] :=   9 ;
  float_sigfigs_per_style [ 2 ] :=   4 ;

                                        (* for miprint *)
  int_ncols_per_style      [ 1 ] :=   10 ;
  int_colwidth_per_style  [ 1 ] :=    7 ;

  int_ncols_per_style      [ 2 ] :=   20 ;
  int_colwidth_per_style  [ 2 ] :=    5 ;

end.
```